Assignment 1: RPCs

Due date: Monday June 15th at 11:00pm Waterloo time

ECE 454/751: Distributed Computing

Instructor: Dr. Wojciech Golab wgolab@uwaterloo.ca

A few house rules

Collaboration:

- groups of 1, 2 or 3 students

Managing source code:

- do keep a backup copy of your code outside of ecelinux, for example using GitLab (https://git.uwaterloo.ca/)
- do not post your code in a public repository (e.g., GitHub free tier)

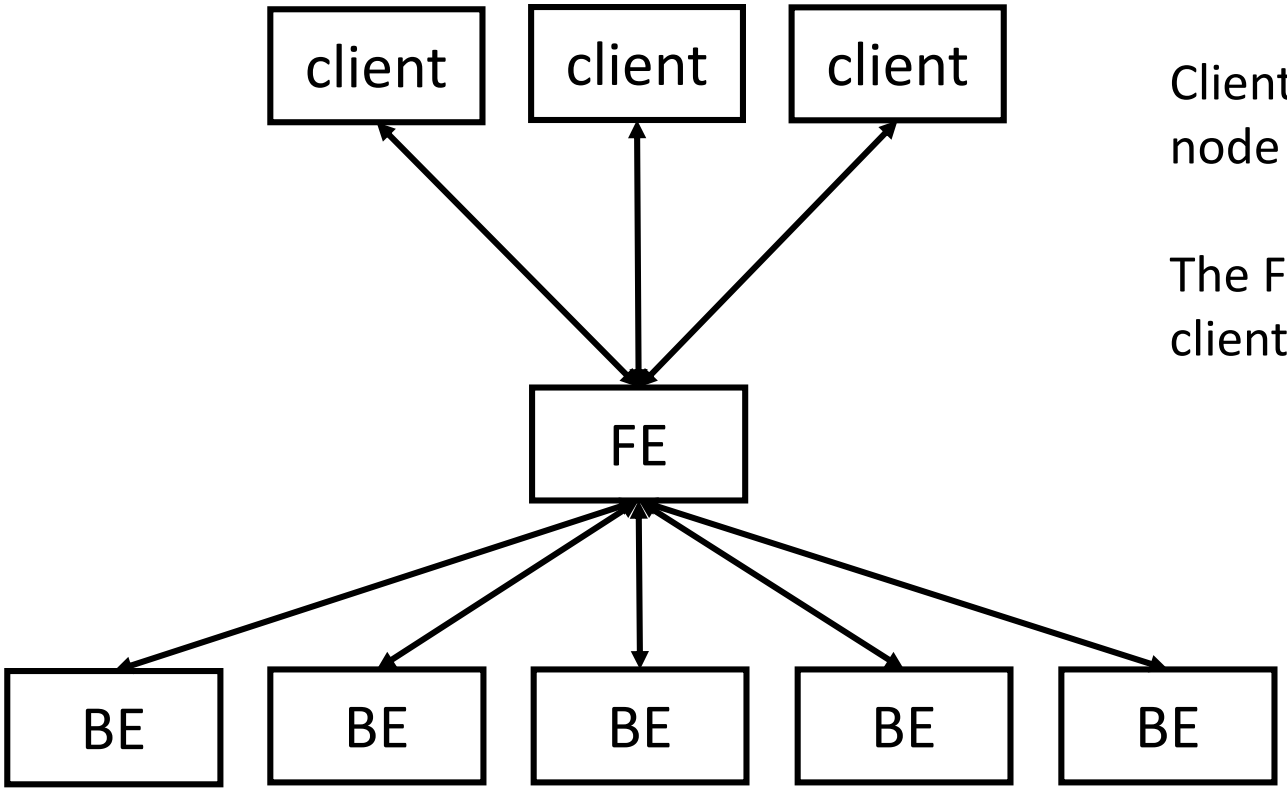
Software environment:

- test on eceubuntu and ecetesla
- use Apache Thrift 0.13.0 and Java 11
- Guava is provided with the starter code
- no other third party code or external libraries are permitted

Overview

- In this assignment you will build a scalable distributed system in Java for computing the bcrypt key derivation function, which is a popular technique for securing passwords in web applications.
- The system will comprise a client layer, a front end (FE) layer, and a back end (BE) layer. The FE layer will accept connections from clients and forward requests to the BE layer in a manner that balances load. The BE layer will be distributed horizontally for scalability, and the FE layer will be centralized for simplicity.
- The objectives of the assignment are:
  1. to gain hands-on experience with Apache Thrift
  2. to learn about scalability and load balancing
- This assignment is worth 20% of your final course grade.

Software architecture



Clients talk to the FE node (and no one else).

The FE node talks to the clients and BE nodes.

Functional requirements

The system supports two fundamental computations on passwords:

- **Hash password:** Hash a given password using a specified number of bcrypt rounds. The output is represented as a string encoding the number of rounds, the cryptographic salt, and the hash.
- **Check password:** check that a given password matches a previously computed hash.

Note 1: We will use jBcrypt-0.4 for cryptographic computations.

Note 2: Passwords will be processed in batches. Thus, the RPCs will accept a list as input and return a list as output. The i'th element of the output list must correspond to the i'th element of the input list. (More detail on the next slide.)

RPC interface for FE layer
(expressed using the Thrift IDL, given in starter code)

```
exception IllegalArgument {
  1: string message;
}

service BcryptService {
 list<string> hashPassword (1: list<string> password, 2: i16 logRounds)
throws (1: IllegalArgument e);
 list<bool> checkPassword (1: list<string> password, 2: list<string> hash)
throws (1: IllegalArgument e);
}
```

Exceptions

The FE node must throw an IllegalArgument exception at the client in the following cases:

- any of the list arguments passed to hashPassword or checkPassword is empty
- the password and hash arguments of checkPassword are lists of unequal length
- the logRounds argument of hashPassword is out of range with respect to the range of values supported by jBcrypt

The FE node must not throw an exception in the following cases:

- empty password passed to hashPassword or checkPassword
- malformed hash passed to checkPassword
  (checkPassword should return false if the hash is malformed)

Code structure

- The starter code provides a partial implementation of the front end and back end nodes in FENode.java and BENode.java, a partial implementation of the service handler in BcryptServiceHandler.java, as well as a primitive client in Client.java.
- The Thrift service definition is provided in a1.thrift.
  Feel free to add procedures to the BcryptService but do not remove anything.
- The jbcrypt.jar is included in jBCrypt-0.4/.
- A shell script called build.sh is provided for compiling the starter code.

Process initialization

- The FE process receives a port number on the command line. It should bind its BcryptService to that port. The FE process is launched as follows:

```
java -cp "..." FENode FE_port
```

- Each BE process receives the host name and port number of the FE process on the command line, as well as a second port number for its own BrcyptService. The BE process is launched as follows:

```
java -cp "..." BENode FE_host FE_port BE_port
```

- **Note:** The correct class path is stated in comments at the bottom of build.sh.

Process initialization: example

1. First launch the FE node:

   java -cp "..." FENode 10123

2. Next, launch a BE node on the same host:

   java -cp "..." BENode localhost 10123 10124

3. Now launch your client.

4. Try again but launch the BE nodes before the FE node. Also test with the client, FE and BEs on different hosts.

Firewall

- Friendly reminder: ports 10000-11000 have been opened on ecelinux hosts to support your assignment.
- Your server processes should only bind to ports in this range, otherwise the firewall may block incoming requests from remote hosts.
- The firewall does not block outgoing connections.
- **Hint:** To reduce the likelihood of port conflicts, try choosing the port number by adding the last three digits of your student number to 10000. It is OK to reuse the same port number on different hosts.

Evaluation

Grading scheme:

Efficiency and scalability:	70%
Fault tolerance:	20%
Exception handling:	10%

A penalty of up to 100% will be applied in the following cases:

- solution cannot be compiled or throws an exception during testing despite receiving valid input
- solution produces incorrect outputs, for example due to a logic error or concurrency bug

Packaging and submission

- Your Java solution comprises FENode.java, BENode.java, and BcryptServiceHandler.java. You may add new .java files but all your Java classes must be in the default Java package. Your Client class will not be graded.
- The solution must use TFramedTransport and TBinaryProtocol (as in the starter code) for compatibility with the grading script.
- Please enter the Nexus IDs of your group members, one member per line, into the group.txt file.
- Use the provided package.sh script to create your tarball for electronic submission, and upload it to the appropriate LEARN dropbox before the deadline.

Group members file

Include in your submission a text file called group.txt that lists the alphanumeric Nexus IDs of your group members, with one group member per line.

Example:

bsimpson
nmuntz
rwiggum

Note: Do not place numerical student IDs or names in this file.

Additional Details

Scalability

- The grading script will use a client with up to twenty threads and will perform synchronous RPCs on your FE node.
- Each client thread will issue requests in a loop using batches of up to 128 passwords at a time, and each password will contain up to 1024 characters. The logRounds parameter will be between 4 and 16.
- There will be one FE node and up to four BE nodes during grading. Each process will run on up to two cores.
- The system must support on-the-fly addition of BE nodes.

Fault tolerance

- The system must tolerate BE node failures. For example, the client should not observe any exceptions if the number of BE nodes is reduced from 2 to 1, or even from 1 to 0. The FE node should perform all the computations locally if no BE nodes are available.
- BE node failures will be simulated using kill -9 (POSIX SIGKILL).
- The system does not need to tolerate FE node failures.
- The system must allow the FE and BE nodes to be launched in either order (e.g., FE before BE, or FE after BE).

Guidelines for performance

- Let's fix the logRounds parameter at some value R.
- Let D denote the time required to perform one cryptographic operation on one core using logRounds=R.
- Let W denote the total number of worker threads at servers (assume this is less than or equal to the total number of cores allocated to the server processes).
- The maximum throughput with twenty client threads, one FE, and four BEs should be close to W/D cryptographic operations per second, especially for larger R (e.g., R=10).
- The latency at the client should be close to D when clients issue requests for one password at a time (i.e., input list has size = one).

Workloads

The grading script will mostly (but not exclusively) test your solution against three workloads:

a) 1 client thread, 16 passwords per request, logRounds from 8 to 12.

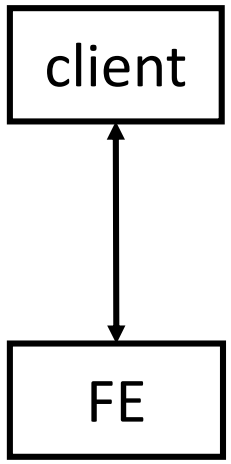
b) 16 client threads, 1 password per request, logRounds from 8 to 12.

c) 4 client threads, 4 passwords per request, logRounds from 8 to 12.

Help, I’m stuck!

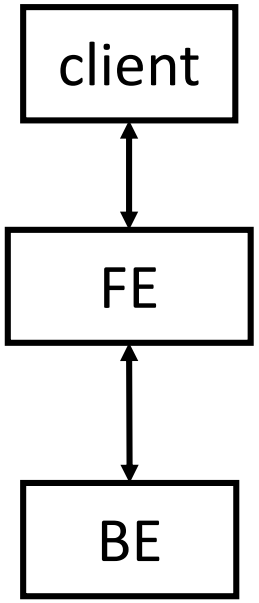
Where do I begin?

- The starter code provides a partial implementation of the BcryptService.
- Try running the provided client and FE node.
- Next, complete the BcryptServiceHandler implementation to handle batches of passwords.



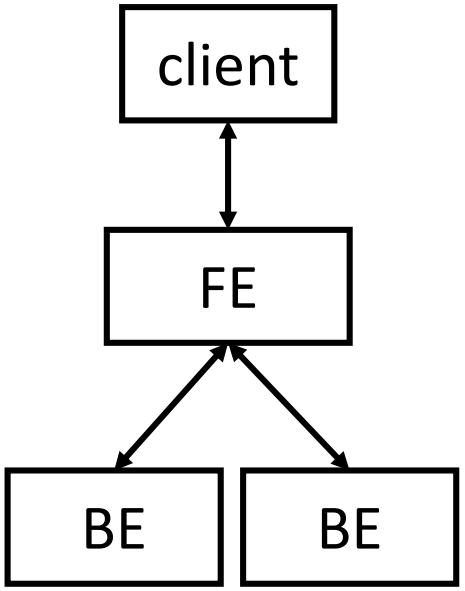
Step 2: add a BE node

- Add some code to the BE node mainline that contacts the FE node on startup.
- Update the FE node so that it offloads some of the work to the BE node if one is available.



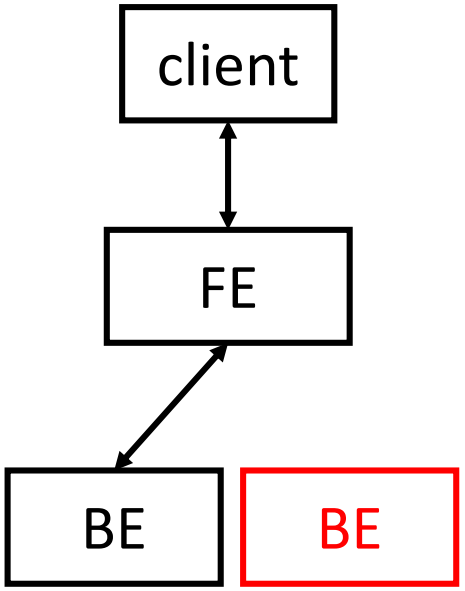
Step 3: load balancing

- Add code to the FE node to support multiple BE nodes.
- Implement and optimize a load balancing strategy.



Step 4: BE node failure

- Add code to the FE node to handle BE node failures.
- If a BE node fails while performing a crypto operation, the work must be reassigned to another node.
- The FE node must handle a BE failure transparently, without throwing an exception at the client.



Step 5: testing and polishing

- Update the code to throw IllegalArgument in the appropriate circumstances.
- Test the implementation thoroughly.
  - Verify that your outputs are always correct.
  - Ensure that the BE nodes can start up successfully without the FE, and vice versa.
  - Inject some BE failures.
  - Use large passwords and large batches in some of your tests.